Hyperbolic Caching: Flexible Caching for Web Applications

Aaron Blankstein, ***Princeton University;*** **Siddhartha Sen,** ***Microsoft Research;***
Michael J. Freedman, ***Princeton University***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

Hyperbolic Caching: Flexible Caching for Web Applications

Aaron Blankstein*, Siddhartha Sen†, and Michael J. Freedman*
* Princeton University, † Microsoft Research

Abstract

Today's web applications rely heavily on caching to reduce latency and backend load, using services like Redis or Memcached that employ inflexible caching algorithms. But the needs of each application vary, and significant performance gains can be achieved with a tailored strategy, e.g., incorporating cost of fetching, expiration time, and so forth. Existing strategies are fundamentally limited, however, because they rely on data structures to maintain a total ordering of the cached items.

Inspired by Redis's use of random sampling for eviction (in lieu of a data structure) and recent theoretical justification for this approach, we design a new caching algorithm for web applications called *hyperbolic caching*. Unlike prior schemes, hyperbolic caching decays item priorities at variable rates and continuously reorders many items at once. By combining random sampling with lazy evaluation of the hyperbolic priority function, we gain complete flexibility in customizing the function. For example, we describe extensions that incorporate item cost, expiration time, and windowing. We also introduce the notion of a *cost class* in order to measure the costs and manipulate the priorities of all items belonging to a related group.

We design a hyperbolic caching variant for several production systems from leading cloud providers. We implement our scheme in Redis and the Django web framework. Using real and simulated traces, we show that hyperbolic caching reduces miss rates by ~10-20% over competitive baselines tailored to the application, and improves end-to-end throughput by ~5-10%.

1 Introduction

Web applications and services aggressively cache data originating from a backing store, in order to reduce both access latency and backend load. The wide adoption of Memcached [23] and Redis [44] (key-value caching), Guava [26] (local object caching), and Varnish [50] (front-end HTTP caching) speak to this demand, as does their point-and-click availability on cloud platforms like Heroku via MemCachier [38], EC2 via ElastiCache [4], and Azure via Azure Redis Cache [7].

Caching performance is determined by the workload and the *caching algorithm*, i.e., the strategy for prioritizing items for eviction when the cache is full. All of the above services employ inflexible caching algorithms, such as LRU. But the needs of each application vary, and significant performance gains can be achieved by tailoring the caching strategy to the application: e.g., incorporating cost of fetching, expiration time, or other factors [8, 46]. Function-based strategies [2, 52] take this approach, by devising functions that combine several of these factors.

All of these strategies are fundamentally limited, however, because they rely on data structures (typically priority queues) to track the ordering of cached items. In particular, an item's priority is only changed when it is accessed. However, does cache eviction need to be tied to a data structure? Caches like Redis already eschew ordering data structures to save memory [45]. Instead, they rely on random sampling to evict the approximately lowest-priority item [42]: a small number of items are sampled from the cache, their priorities are evaluated (based on per-item metadata), and the item with lowest priority is evicted. Can this lack of an ordering data structure enable us to build a caching framework with vast flexibility? Indeed, we show that the combination of *random sampling* and *lazy evaluation* allows us to evolve item priorities arbitrarily; thus we can freely explore the design space of priority functions! Neither Redis nor existing algorithms exploit this approach, yet we find it outperforms many traditional and even domain-optimized algorithms.

Armed with this flexibility, we systematically design a new caching algorithm for modern web applications, called *hyperbolic caching* (§2). We begin with a simple theoretical model for web workloads that leads to an optimal solution based on frequency. A key intuition behind our approach is that caches can scalably measure item frequency *only while items are in the cache*. (While some algorithms, e.g., ARC [37], employ *ghost caches* to track items not in the cache, we focus on the more practical setting where state is maintained only for cached items.) Thus, we overcome the drawbacks of prior frequency-based algorithms by incorporating the time an item spends in the cache. This deceptively simple modification already makes it infeasible to use an ordering data structure, as pervasively employed today, because item priorities decay at variable rates and are continuously being reordered. Yet with hyperbolic caching, we can easily customize the priority function to different scenarios by adding *extensions*, e.g., for item cost, expiration time, and windowing (§3). We also introduce the notion of *cost classes* to man-

age groups of related items, e.g., items materialized by the same database query. Classes enable us both to more accurately measure an item's miss cost (by averaging over multiple items) and to adjust the priorities of many items at once (e.g., in response to a database overload).

A quick survey of existing algorithms shows that they fall short of this flexibility in different ways. Recency-based algorithms like LRU use time-of-access to order items, which is difficult to extend: for example, incorporating costs requires a completely new design (e.g., GreedyDual [53]). Frequency-based algorithms like LFU are easier to modify, but any non-local change to item priorities—e.g., changing the cost of multiple items—causes expensive churn in the underlying data structure. Some algorithms, such as those based on marking [22], maintain only a partial ordering, but the coarse resolution makes it harder to incorporate new factors. Several theoretical studies [2,46] formulate caching as an optimization problem unconstrained by any data structure, but their solutions are approximated by online heuristics that, once again, rely on data structures.

We design a hyperbolic caching variant for several different production systems from leading cloud providers (§3), and evaluate them on real traces from those systems. We implement hyperbolic caching in Redis and the Django web framework [18], supporting both per-item costs and cost classes (§4). Overall (§5), we find that hyperbolic caching reduces miss rates by ~10-20% over competitive baselines tailored to the application, and improves end-to-end system throughput by ~5-10%. This improvement arises from changing only the caching algorithms used by existing systems—our modification to Redis was 380 lines of code—and nothing else.

To summarize, we make the following contributions:

1. We systematically design a new caching algorithm for modern web applications, hyperbolic caching, that prioritizes items in a radically different way.
2. We define extensions for incorporating item cost and expiration time, among others, and use them to customize hyperbolic caching to three production systems.
3. We introduce the notion of cost classes to manage groups of related items effectively.
4. We implement hyperbolic caching in Redis and Django and demonstrate performance improvements for several applications.

Although we only evaluate medium-to-large web applications, we believe hyperbolic caching can improve hyper-scale applications like Facebook, where working sets are still too large to fit in the cache [6, 49].

2 Hyperbolic Caching

We first describe the caching framework required by hyperbolic caching (§2.1). Then, we motivate a simple theoretical model for web workloads and show that a classical frequency approach is optimal in this model (§2.2). By solving a fundamental challenge of frequency-based caching (§2.3), we arrive at hyperbolic caching (§2.4).

2.1 Framework

We assume a caching service that supports a standard `get`/`put` interface. We make two changes to the implementation of this interface. First, we store a small amount of metadata per cached item i (e.g., total number of accesses) and update it during accesses; this is done by the `on_get` and `on_put` methods in Fig. 1. Second, we remove any data structure code that was previously used to order the items. We replace this with a priority function `p(i)` that maps item i's metadata to a real number; thus `p` imposes a total ordering on the items. To evict an item, we randomly sample S items from the cache and evict the item i with lowest priority `p(i)`, as implemented by `evict_which`. This approximates the lowest-priority item [42]; we evaluate its accuracy in §5.3.

The above framework is readily supported by Redis, which already avoids ordering data structures and uses random sampling for eviction. The use of metadata and a priority function is standard in the literature and referred to as "function-based" caching [8]. What is different about our framework is *when* this function is evaluated. Prior schemes [2, 46, 52] evaluate the function on each `get`/`put` and use the result to (re)insert the item into a data structure, freezing its priority until subsequent accesses. Our framework uses *lazy evaluation* and no data structure: an item's priority is only evaluated when it is considered for eviction, and it can evolve arbitrarily before that point without any impact on performance.

2.2 Model and frequency-based optimality

In many workloads, the requests follow an item popularity distribution and the time between requests for the same item are nearly independent [10]. Absent real data, most systems papers analyze such distributions (e.g., Zipfian [20, 56]), and model dynamism as gradual shifts between static distributions. Motivated by this, we model requests as a *sequence of static distributions* $\langle D_1, D_2, \ldots \rangle$ over a universe of items, where requests are drawn independently from D_1 for some period of time, then from D_2, and so on. The model can be refined by constraining the transitions (D_i, D_{i+1}), but even if we assume they are instantaneous, we can still prove some useful facts (summarized below). Our measure of cost is the miss rate, which is widely used in practice.

```
def evict_which():
  sampled_items = random_sample(S)
  return argmin(p(i) for i in sampled_items)

def on_put(item):
  item.accessed = 1
  item.ins_time = timenow()
  add_to_sampler(item)

def on_get(item):
  item.accessed += 1

def p(item):
  in_cache = timenow - item.ins_time
  return item.accessed / (in_cache)
```

Figure 1: Pseudocode for hyperbolic caching in our framework.

Within a distribution D_i, a simple application of the law of large numbers shows that the optimal strategy for a cache of size k is to cache the k most popular items. This is closely approximated by the least-frequently-used (LFU) algorithm: a typical implementation assigns priority n_i/H to item i, where n_i is the number of hits to i and $H = \sum_i n_i$ is the sum over all cached items. Whereas LFU approximates the optimal strategy, one can prove that LRU suffers a gap. This is in contrast to the traditional competitive analysis model—which assumes a *worst-case* request sequence and use *total misses* as the cost [47]—in which LRU is optimal. This model has been widely criticized (and improved upon) for being pessimistic and unrealistic [3, 9, 32, 33, 54]. Our model is reminiscent of older work (e.g., [24]) that studied independent draws from a distribution but, again, used total misses as the cost.

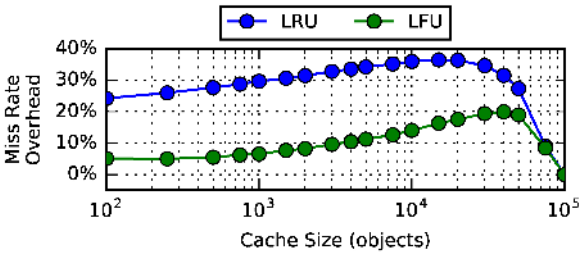
To validate our theoretical results, we use a static Zipfian popularity distribution and compare the miss rates of LRU and LFU to the optimal strategy, which has perfect knowledge of every item's popularity (Fig. 2).[1] Until the cache size increases to hold most of the universe of items, LRU has a 25-35% higher miss rate than optimal. LFU fares considerably better, but is far from perfect. We address the drawbacks of LFU next.

2.3 Problems with frequency

Even if requests are drawn from a stable distribution, there will be irregularities in practice that cause well-known problems for frequency-based algorithms:

New items die. When an item is inserted into the cache, the algorithm does not have a good measure of it's popularity. In LFU, a new item gets a frequency count of 1, and may not have enough time to build up its count to survive in the cache. In the worst case, it could be repeatedly inserted and evicted despite being requested frequently.

[1]We present miss rate rather than hit rate curves because our focus is on the penalties at the backend. Higher numbers indicate *worse* performance in most figures, and the last datapoint is 0 because the cache is large enough to never incur a miss.



Cache Size	3k	10k	30k	100k
Perfect Freq. Miss Rate	0.29	0.19	0.10	0.00

Figure 2: Simulated miss rates[1] compared to a strategy with perfect frequency knowledge. Items are sampled with Zipfian popularity ($\alpha \approx 1$) from 10^5 items. The cache is configured to hold a fixed number of objects (rather than simulating size in bytes).

This problem can be mitigated by storing metadata for non-cached items (e.g., [37]), but at the cost of additional memory that is worst-case linear in the universe size.

Old items persist. When items' relative popularities shift—e.g., moving from D_i to D_{i+1} in our model—a frequency approach may take time to correct its frequency estimates. This results in older items persisting in the cache for longer than their current popularity warrants. For example, consider a new item with 1 access and an older item with 2 accesses. Initially, the new item may be better to cache, but if time passes without an additional access, our knowledge of the old item is more reliable.

2.4 Hyperbolic Caching

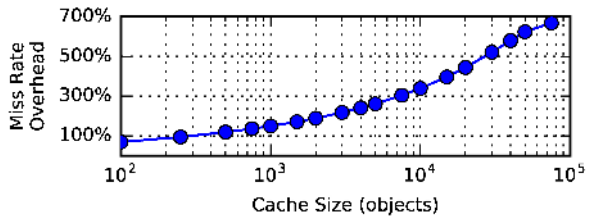
We solve the above problems by incorporating a *per-item* notion of time. Intuitively, we want to compensate for the fact that caches can only measure the frequency of an item *while it is in the cache*. Traditional LFU does not account for this, and thus overly punishes new items.

In our approach, an item's priority is an estimate of its frequency since it entered the cache:

$$p_i = \frac{n_i}{t_i} \tag{1}$$

where n_i is the request count for i since it entered the cache and t_i is the time since it entered the cache. This state is erased when i is evicted. Fig. 1 provides pseudocode for this policy, which we call *hyperbolic caching*.

Hyperbolic caching allows a new item's priority to converge to its true popularity from an initially high estimate. This initial estimate gives the item temporary immunity (similar to LRU), while allowing the algorithm to improve its estimate of the item's popularity. Over time, the priority of each item drops along a hyperbolic curve. Since each curve is unique, the ordering of the items is continuously changing. Such reordering is uniquely enabled by our framework (lazy evaluation, random sampling), and



Cache Size	3k	10k	30k	100k
HC Miss Rate	0.30	0.21	0.13	0.00

Figure 3: LFU miss rate compared to hyperbolic caching (HC) on a dynamic Zipfian workload ($\alpha \approx 1$), where new items are introduced into the distribution every 100 requests.

would be very costly to implement with a data structure.[2]

The strengths of hyperbolic caching over LFU are readily apparent in workloads that slowly introduce new items into the request pool. Fig. 3 shows that LFU has a significantly higher miss rate on a workload that introduces new items every 100 requests whose popularities are in the top 10% of a Zipfian distribution. This workload is artificial and much more dynamic than we would expect in practice, but serves to illustrate the difference.

Another way to solve the same problem is to multiplicatively degrade item priorities (e.g., LRFU [34]) or periodically reset them. Both of these are forms of windowing, which best addresses the problem of old items persisting, *not* the problem of new items dying. We compare hyperbolic caching to these approaches in §3.4.

3 Customizing Hyperbolic Caching

Our framework allows us to build on the basic hyperbolic caching scheme by adding *extensions* to the priority function and storing metadata needed by those extensions. This is similar to the way function-based policies build on schemes like LRU and LFU [2, 46, 52], but in our case the extensions can freely modify item priorities without affecting efficiency (beyond the overhead of evaluating the function). Which extensions to use and how to combine them are important questions that depend on the application. Here, we describe several extensions that have benefited our production applications (cost, expiration time) and our understanding of hyperbolic caching's performance (windowing, initial priority estimates).

3.1 Cost-aware caching

In cost-aware caching, all items have an associated cost that reflects the penalty for a miss on the item. The goal is to minimize the total cost of all misses. Cost awareness is particularly relevant in web applications, because unlike traditional OS uses of caching (fixed-size CPU instruction lines, disk blocks, etc.), the cost of fetching different items can vary greatly: items vary in size, can originate from different backing systems or stores, or can be the materialized result of complex database joins.

Much of the prior work on cost-aware caching focuses on adapting recency-based strategies to cost settings (e.g., GreedyDual [11]). This typically requires a new design, because recency-based strategies like LRU-K [41] and ARC [37] use implicit priorities (e.g., position in a linked list) and metrics like time-of-access, which are difficult to augment with cost. In contrast, frequency-based approaches like hyperbolic caching use explicit priorities that can naturally be multiplied by a cost: $p_i' = c_i p_i$, where c_i is the cost of fetching item i and p_i is the original (cost-oblivious) priority of i. Note that p_i may include other extensions from later sections.

The cost of an item needs to be supplied to the caching algorithm by the application. It can take many forms. For example, if the goal is to limit load on a backing database [35], the cost could be request latency. If the goal is to optimize the hit rate per byte of cache space used, the cost could be item size [11].

Real-world applications. Our evaluation studies two applications which benefit from cost awareness. The first is a set of applications using Memcachier [38], a production cloud-based caching service built on Memcache. We use costs to account for object size in the eviction decision, i.e., set $c_i = 1/s_i$ where s_i is the size of item i. The second application is Viral Search [25, 51], a Microsoft internal website that displays viral stories from Twitter in tree form. Virality is measured by analyzing the diffusion tree of the story as it is shared through the network. For each story, the website fetches the tree edges and constructs and lays them out for display. The final trees are cached and the cost of each is set to the time required to construct and lay out the tree.

3.2 Cost classes

In many applications, the costs of items are related to one another. For example, some items may be created by database joins, while others are the result of simple indexed lookups. Rather than measuring the cost of each item individually, we can associate items with a *cost class* and measure the performance of each class. We store a reference to the class in each item's metadata.

Cost classes have two main advantages. Consider the example of request latency to a backend database. If costs are maintained per item, latencies must be measured for each insertion into the cache. Since these measurements are stochastic, some requests will experience longer de-

[2]The basic hyperbolic function in Eq. 1 can be tracked by a kinetic heap [31], but this is a non-standard structure with $O(\log^2 n)$ update time, and it ceases to work if the extensions from §3 are added.

lays than others and thus be treated as more costly by the cache, even though the higher cost has nothing to do with the item itself. What's more, the higher costs will keep these items in the cache longer, preventing further updates because costs are only measured when a miss occurs. By using cost classes, we can aggregate latency measurements across all items of a class (e.g., in a weighted moving average), resulting in a less noisy estimate of cost.

The second advantage of cost classes comes from longer-term changes to costs. In scenarios where a replica failure or workload change affects the cost of fetching a whole class of items, existing approaches would only update the individual costs after the items have been evicted, one by one. However, when using cost classes, a change to a class's cost is immediately applied to both newly cached items and items *already* in the cache.

In both cases above, a single update to a cost class changes the priorities of many items at once, possibly dramatically. Our framework supports this with little additional overhead because 1) items store a *reference* to the class information, and 2) priorities are lazily evaluated. In contrast, integrating cost classes into existing caching schemes is prohibitively expensive because it incurs widespread churn in the data structures they rely on.

Interestingly, some production systems already employ cost classes implicitly, via more inflexible and inelastic means. For example at Facebook, the Memcached service is split among a variety of pools, such that keys that are accessed frequently but for which a miss is cheap do not interfere with infrequently accessed keys for which a miss is very expensive [40]. However, this scheme requires much more management and requires tuning pool sizes; more importantly, it does not automatically adapt to changes in request frequencies or item costs.

In our experiments, we implement cost classes using exponentially weighted moving averages. We explored other techniques such as non-weighted moving averages and using the most recent cost, but exponentially weighted moving averages performed the best on our workloads while requiring little memory overhead for tracking.

While cost classes are useful in many settings, incorrectly assigning objects to the same class that do not share the same underlying cost will degrade caching performance. In some settings, objects may be members of multiple classes concurrently—there are several ways of handling this, but we do not explore this in our work.

Real-world application. Django is a Python framework for web apps that includes a variety of libraries and components. One such component adds support for whole-page caching. We modified this middleware to support cost awareness, as follows. In Django, page requests are dispatched to "view" functions based on the URL. We associate a cost class with each view function, and map individual pages to their view function's class.

3.3 Expiration-aware caching

Many applications need to ensure that the content conveyed to end users is not stale. Developers achieve this by specifying an *expiration time* for each item, which tells the caching system how long the item remains valid. While many systems support this feature, it is typically handled by an auxiliary process that has no connection to the caching algorithm (apart from evicting already-expired items). But incorporating expiration into caching decisions makes intuitive sense: if an item is going to expire soon, it is less costly to evict than a similarly popular item that expires later (or not at all).

To add expiration awareness to hyperbolic caching, we need to strike a balance between the original priority of an item and the time before it expires. Rather than evict the item least likely to be requested *next*, we want to evict the item most likely to be requested *the least number of times over its lifetime*. This can be naturally captured by multiplying item i's priority by the time remaining until expiry, or $\max((t_{exp_i} - t_{cur}), 0)$. However, this scheme equally prioritizes requests far into the future and those closer to the present, which is unideal because estimates about the future are less likely to be accurate (e.g., the item's popularity may change). Therefore, instead of equally weighting all requests over time, we use a weighting function that discounts the value of future requests:

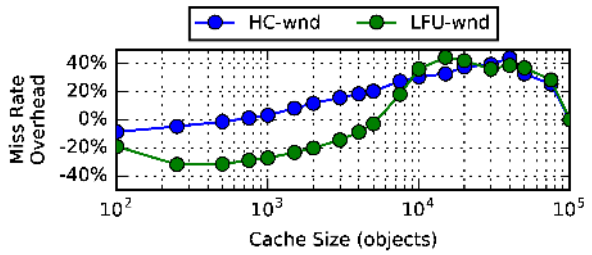
$$p_i' = p_i \cdot (1 - e^{-\lambda \cdot \max((t_{exp_i} - t_{cur}), 0)})$$

where p_i is the original (expiration-unaware) priority of item i and λ is a parameter controlling how quickly to degrade the value of future requests. As an item's time until expiration decreases, this weighting function sharply approaches zero. Thus the function continually reweights (reorders) item priorities, which is uniquely enabled by our framework: existing approaches can only account for expiration time once, on insertion into a data structure.

Real-world application. The Decision Service [1,39] is a machine learning system for optimizing decisions that has been deployed in MSN to personalize news articles shown to users. Given a user request, a particular article is featured and a reward signal (e.g., click) is recorded. Since rewards may arrive after a substantial delay, a cache is used to match the decision to its reward. Rewards are only valid if they occur within a time window after the decision, so each cached item is given an expiration time.

3.4 Windowing

Windowing is often used in frequency-based caching to adapt to dynamic workloads and address the problem of



Cache Size	3k	10k	30k	100k
HC Miss Rate	0.32	0.22	0.13	0.00
LFU Miss Rate	0.44	0.21	0.13	0.00

Figure 4: Adding perfect windowing to hyperbolic caching and LFU on a dynamic Zipfian workload ($\alpha \approx 1$). Each curve is compared to the algorithm's non-windowed performance (given in the table). The window size is fixed at 10^4 requests. Every 100 requests, an item is promoted to the top of the distribution.

"old items persisting". The idea is to forget requests older than a fixed time window from the present. Hyperbolic caching naturally achieves the benefits of windowing, but we investigate it for two reasons. First, one can show that hyperbolic caching, unlike LRU, is not optimal in the traditional competitive analysis model [47], but it can be made optimal if windowing is used. Second, windowing represents alternative solutions, such as resetting or multiplicatively degrading frequency estimates (e.g., LRFU [34]), and so serves as an informative comparison.

We simulate windowing using an idealized (but completely inefficient) scheme that tracks every request and forgets those older than the window. This upper bounds the potential gains of windowing. Fig. 4 shows the performance of LFU and hyperbolic caching on a dynamic Zipfian workload, with and without windowing. For hyperbolic caching, windowing provides limited benefits: 5–10% reduction in misses on small cache sizes; LFU benefits more but again on small cache sizes. The problem is that windowing discards measurements that help the cache estimate item popularity. Even in dynamic workloads, we find that large-sized caches can accommodate newly popular items, so performance depends more on the ability to differentiate at the long tail of old items. Fortunately, hyperbolic caching's measure of time in the cache achieves some of the benefits of windowing; it outperforms even recency-based approaches on many of the highly dynamic workloads we evaluated.

3.5 Initial priorities

Hyperbolic caching protects newly cached items by giving them an initial priority that tends to be an *overestimate*: for example, an item with true popularity of 1%—placing it among the most popular in most realistic workloads—would remain overvalued for at least 100 timesteps of hyperbolic decay. We found that adjusting the initial priority based on that of recently evicted items alleviates this problem, because evicted items tend to have similar priorities in the tail of the distribution. Thus, we set a new item's initial priority to a mixture of its original priority (p_i) and the last evicted item's priority (p_e): $p_i' = \beta p_i + (1-\beta)p_e$. Solving this for n_i in Eq. 1 gives us the initial request count to use, after which the extension can be discarded. β requires some tuning: we found that $\beta = 0.1$ works well on many different workloads; for example, on a Zipfian workload ($\alpha \approx 1$) it reduced the miss rate by between 1% and 10% over hyperbolic caching for all cache sizes.

4 Implementation

Our evaluation uses both simulation and a prototype implementation. For the simulations, we developed a Python application that generates miss rate curves for different caching strategies and workloads. For our prototype, we implemented hyperbolic caching in Redis and developed Django middleware that uses the modified Redis. Our code is open-source [28].

Redis. We modified Redis (forked at 3.0) to use the hyperbolic caching framework. This was straightforward because Redis already uses random sampling for eviction. We included support for per-item costs (and size awareness), cost classes tracked with an exponentially weighted moving average, and initial priorities. Excluding diagnostic code, this required 380 lines of C code.

We store the following metadata per item, using double-precision fields: item cost, request count, and time of entry (from Eq. 1 and §3.1). This is two doubles of overhead per item compared to LRU. Our prototype achieved similar miss rates to our simulations, suggesting this precision is adequate. Exploring the trade-offs of reduced precision in these fields is left to future work.

Django caching middleware. Django is a framework for developing Python web applications. It includes support for middleware classes that enable various functionality, such as the Django whole-page caching middleware. This middleware interposes on requests, checking a backend cache to see whether a page is cached, and if so, the content is returned to the client. Otherwise, page processing continues as usual, except that the rendered page is cached before returning to the client. We added middleware to track cost information for web pages; we measure cost as the CPU time between the initial miss for a page and the subsequent SET operation, plus the total time for database queries. This avoids time lost due to processor scheduling. We subclassed the Django Redis caching interface to convey cost information to our Redis implementation. The interface supports caching a page

with/without costs, and optionally specifying a cost class for the former. Cost classes are associated with the particular Django "view" function that renders the page. In total, this was implemented in 127 lines of Python code.

5 Evaluation

Our evaluation explores the following questions:

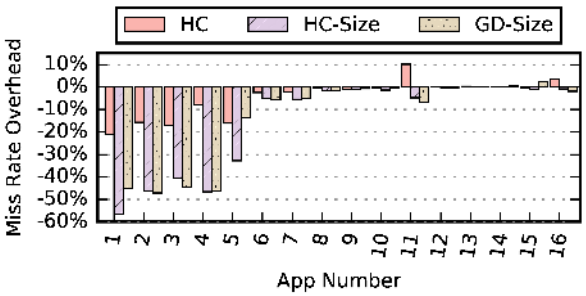
1. How does hyperbolic caching compare to current caching techniques in terms of miss rate?
2. Does our implementation of hyperbolic caching in Redis improve the throughput of web applications?
3. What effect does sample size have on the accuracy and performance of our eviction strategy?

We use real application traces (§5.1) and synthetic workloads designed to emulate realistic scenarios (§5.2). We evaluate these questions using simulations as well as deployments of Django and NodeJS, using our prototype of hyperbolic caching in Redis. To drive our tests, our applications run on Ubuntu 14.04 servers located on a single rack with Intel Xeon E5620 2.40GHz CPUs. Applications use PostgreSQL 9.3 as their backing database. For throughput tests, our systems were loaded exclusively by the test, and to measure max throughput, we increased the rate of client requests until throughput plateaued and the application server experienced 100% CPU load.

Methodology. For the majority of our standard workloads, we use a Zipfian request distribution with $\alpha \approx 1$. This is the same parameterization as many well-studied benchmarks (e.g., YCSB [15]), though some like linkbench [5] use a heavier-tailed $\alpha = 0.9$. When measuring miss rates, we tally misses after the first eviction (i.e., we allow the cache to fill first). For workloads with associated item costs, misses are scaled by cost. For real traces, we run the tests exactly as prescribed; for workloads based on popularity distributions, we generate enough requests to measure the steady state performance. When choosing a cache size to compare performance amongst algorithms, we use the size given by the trace, or if not given we use sizes corresponding to high and middle range hit rates (roughly 90% and 70%), which reflect the cache hit rates reported in many deployed settings (e.g, [6, 27]). In Facebook [27], of the 35.5% of requests that leave a client's browser (the rest are cached locally), ~70% are cached in either the edge cache or the origin cache. For our random sampling, unless otherwise noted, we sample 64 items.

5.1 Real-world workloads

We evaluate real applications in two ways. When lacking access to the actual application code or deployment setting, we evaluate the performance through simulation. For other applications, we measure the performance using our prototype implementation of Django caching paired with Redis. The applications below were described in §3, when we customized hyperbolic caching to each one.



(a) Simulated miss rates compared to the miss rate of LRU.

App Number	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Mean Obj. Sz. (kB)	79.9	15.4	1.8	149.7	561.7	2.3	1.1	25.0
Stdev Obj. Sz. (kB)	116.5	40.4	9.0	254.5	100.8	3.1	7.9	46.6

App Number	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Mean Obj. Sz. (kB)	5.4	8.1	7.2	5.5	2.2	30.8	26.0	9.0
Stdev Obj. Sz. (kB)	5.1	13.5	17.9	2.0	4.3	52.2	3.1	25.4

(b) Means and stdevs. of object sizes in app traces.

Figure 5: Caching performance on Memcachier app traces.

5.1.1 Memcachier applications (from §3.1)

To evaluate the Memcachier applications, we processed a trace of GET and SET requests spanning hundreds of applications, using the amount of memory allocated by each application as the simulated cache size. We focused our attention on the 16 applications with over 10k requests whose allocation could not fit all of the requested objects (many applications allocated enough memory to avoid any evictions). We measured the miss rates of plain HC and LRU, and then used the object sizes to evaluate our size-aware extension, HC-Size, and the GD-Size [11] algorithm. Fig. 5 show the performance of the algorithms over a single execution of each application's trace.

In our evaluation, HC outperforms LRU in many applications, and HC-Size drastically outperforms LRU. While GD-Size is competitive with HC-Size, our framework allows for the implementation of HC-Size with only two lines of code, whereas implementing GD-Size from LRU requires an entirely new data structure [11].

5.1.2 Decision Service (from §3.3)

The Decision Service [1, 39] is a machine learning system for optimizing decisions that has been deployed in MSN. The service uses a cache to join information about each decision with the corresponding reward signal. Because rewards must be received within a given period of time, information is cached with an expiration time.

	Decision Service	Viral Search
Algo.	Miss Rate (Δ%)	Miss Rate (Δ%)
HC	0.60 (+0%)	0.17 (+0%)
HC-Expire	0.55 (-8%)	—
LRU/GD	0.55 (-8%)	0.18 (+6%)
ARC	0.55 (-8%)	0.22 (+29%)
LFU	0.99 (+65%)	0.16 (-6%)
Cache Size	1k	35k

Figure 6: Simulated performance on real-world traces.

Cache Alg.	Miss Rate (Δ)	Tput. (Δ)
Default	0.681 (+0.0%)	21.1 req/s (+0.0%)
HC	0.637 (-6.5%)	23.7 req/s (+12.3%)
HC-Cost	0.669 (-1.8%)	21.1 req/s (+0.0%)
HC-Class	0.639 (-6.2%)	22.6 req/s (+7.1%)
Obj. Sizes	$\mu = 32.0$kB	$\sigma = 31.6$kB

Figure 7: Performance of Django-Wiki application using HC Redis compared to default Redis. The cache is sized to 1GB and the workload is a 600k trace of Wikipedia article requests.

In this workload, because items have the same expiration time and are accessed only once after insertion (to join the reward information), recency is roughly equal to time-until-expiration. Therefore, LFU and HC perform poorly in comparison to a recency strategy (Fig. 6). However, our expiration-aware extension allows HC-Expire to perform just as well as the recency strategies.

5.1.3 Viral Search (from §3.1)

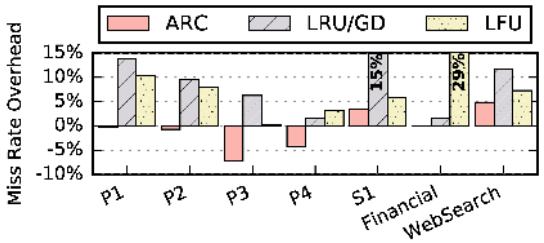
The Viral Search [25,51] application is an interactive website that displays viral stories from a large social network. Each viral story is represented as a tree that requires varying amounts of time to construct and layout on the server side. We use this time as the per-item cost and apply our cost-aware extension. Items are requested based on a popularity distribution given by each item's "virality score" and we measure performance over 10M requests.

Hyperbolic caching performs well on this cost-aware workload, beating all algorithms except for LFU (Fig. 6), and suffering 6% fewer misses than GreedyDual.

5.1.4 Django Wiki application (from §3.2)

We evaluate our caching scheme on an open-source Django wiki app using our Django caching middleware. The caching middleware stores cached data using a configurable backend, for which we use either the default Redis or our modified version with hyperbolic caching.

The wiki database serves a full copy of articles on Wikipedia from Jan. 2008. We measured the throughput and miss rate of the application using a trace of Wikipedia article requests from Sept. 1, 2007 (Fig. 7). We see an improvement in both miss rate and throughput when using HC rather than default Redis. Note that because the pages are costly to render, even small improvements in miss rate increase the throughput of the application. For this application, requests are only processed by two different Django views.



Trace	P1	P2	P3	P4	S1	F	WS
Cache Sz (*objs*)	32k	32k	32k	32k	525k	32k	525k
Miss Rate	0.72	0.73	0.88	0.91	0.81	0.50	0.85

Figure 8: Miss rates compared to HC on traces from the ARC paper and SPC (HC's miss rates are in the table). Cache sizes chosen based on sizes given in the ARC paper.

However, using HC-Cost *reduces* the system throughput compared to HC. This is because the time to render a page is similar across most pages, but has high variance: for one page, the mean time of fifty requests was 570ms with a deviation of 180ms. This leads a cost-aware strategy to incorrectly favor some pages over others. HC-Class alleviates this by reducing some of the variance, but it still performs worse than the cost-oblivious HC. For this application, using costs is counter-productive.

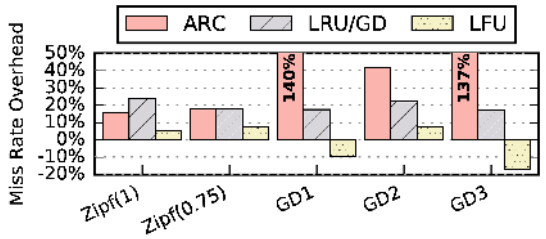
5.1.5 ARC and SPC traces

We additionally simulate performance on traces from ARC [37] and SPC [48] (Fig. 8). The P1-4 traces are memory accesses from a workstation computer; S1 and WebSearch are from a server handling web searches; and the Financial workload is an OLTP system trace. Caches were sized according to the ARC paper, and these sizes were used for the SPC traces as well. These traces have very high miss rates on all eviction strategies. However, HC performs very well, outperforming LRU in every workload and underperforming ARC in the P1-4 traces only. Importantly, on workloads where LFU exhibits poor performance, HC remains competitive with ARC, demonstrating the effectiveness of our improvements over LFU.

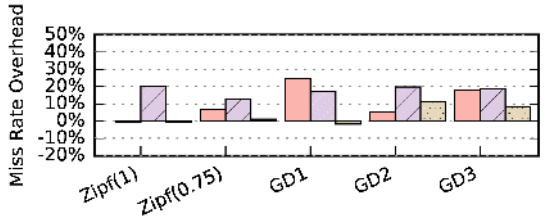
5.2 Synthetic workloads

In this section, we simulate and compare the performance of HC to three popular strategies—ARC, LFU, and LRU—on synthetic workloads that reflect the demands of today's caches. For cost-aware workloads, we extend LRU with GreedyDual, and we modify LFU by multiplying frequencies by cost. (ARC is not amenable to costs.)

For each synthetic workload, we evaluate the performance of each caching algorithm on two cache sizes, corresponding to a 90% and a 70% hit rate with hyper-



(a) Cache size fixed where hit rate of HC $\approx 90\%$.



(b) Cache size fixed where hit rate of HC $\approx 70\%$.

Figure 9: Miss rates on synthetic workloads with 10M requests. Miss rates are compared to the performance of HC. For cost-aware strategies (GD1-GD3), misses are scaled by the cost of the missed item.

bolic caching (Fig. 9). Note that while we simulated relatively small key spaces, we evaluated our Redis prototype on larger key spaces and found similar improvements in miss rate and overall system throughput. In general, these workloads suggest that HC can perform very well in a variety of scenarios.

The most striking improvement relative to ARC is on workloads GD1-3. These workloads have associated costs and are based on the workloads described in GDWheel [35]. Since ARC is a cost-oblivious strategy, it does poorly on these workloads. However, even in workloads without cost, our scheme is competitive with ARC.

5.2.1 Synthetic web application performance.

In order to understand how our improved miss rates affect end-to-end throughput in modern web servers, we configured a NodeJS web app to use a backing database with Redis as a look-aside cache. We drive HTTP GET requests to the web app from a client that draws from synthetic distributions. The web app parses the URL and returns the requested object. Objects are stored as random 32B strings in a table with object identifier as the primary key.

Relating cache misses to throughput. To understand the association between miss rate and throughput, we scaled the size of our Redis cache to measure system throughput with different miss rates (Fig. 10). Miss rate has a direct impact on throughput even when many client requests can be handled concurrently. Misses not only cause slower responses from the backend (an effect which can be mitigated with asynchronous processing), but they also require additional processing on the web server—on a miss, the app issues a failed GET, a SQL SELECT, and then a PUT request. This adds a direct overhead to the throughput of the system.

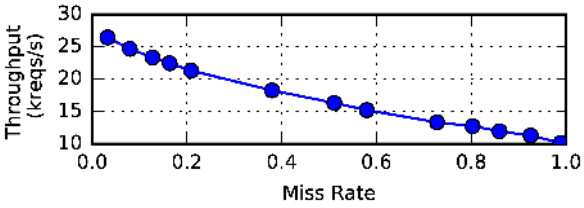


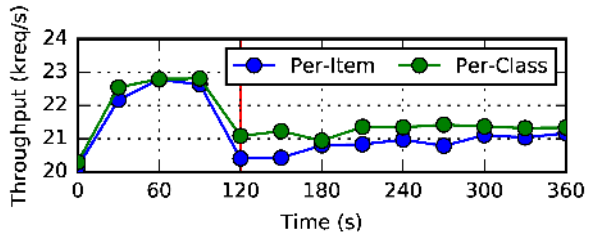
Figure 10: Throughput of NodeJS using Redis as a look-aside cache for PostgreSQL as the miss rate varies.

Cache sz. *(objs.)*	Default Redis Mean tput. *(kreq/s)*	Miss rate	HC Redis Mean tput. *(kreq/s)*	Miss rate	Δ tput.
Zipfian ($\alpha \approx 1, N = 10^5$)					
39k	18.1 ± 0.22	0.11	20.2 ± 0.18	0.09	10.3%
3k	9.1 ± 0.09	0.38	10.5 ± 0.06	0.31	13.5%
Zipfian ($\alpha = 0.75, N = 10^6$)					
125k	7.5 ± 0.06	0.55	7.7 ± 0.16	0.49	3.2%
70k	6.8 ± 0.06	0.64	7.3 ± 0.12	0.56	6.3%
Zipfian ($\alpha \approx 1, N = 10^6$)					
200k	14.6 ± 0.16	0.17	15.3 ± 0.13	0.16	4.4%
50k	11.2 ± 0.11	0.28	12.1 ± 0.20	0.24	7.1%
Dynamic Intro. ($N = 10^5$)					
42k	19.3 ± 0.17	0.10	20.6 ± 0.16	0.09	6.3%
5k	10.0 ± 0.15	0.33	11.3 ± 0.12	0.27	11.6%

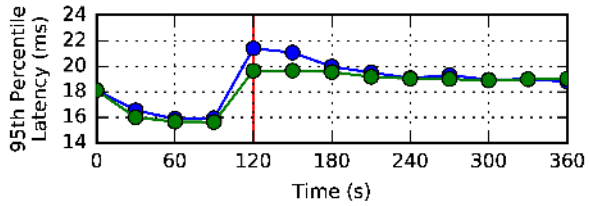
Figure 11: Miss rate and throughput of workloads running on NodeJS with a Redis cache. Each configuration was executed 10 times with workloads of 5M requests to objects of size 96B.

Zipfian distribution. We measured the maximum throughput of our NodeJS server when servicing requests sampled from synthetic workloads with zipfian request distributions (Fig. 11.) Depending on the workload, hyperbolic caching outperforms Redis's default caching algorithm (LRU approximated by random sampling) in miss rates by 10-37%, and improves throughput by up to 14% on some workloads. While throughput differences of 5-10% on some workloads may be modest, they are not insignificant, and come with little implementation burden.

Cost-aware caching. To measure the potential throughput benefits of cost-aware caching, we wrote a NodeJS app that makes two types of queries to the backend: (1) a simple key lookup and (2) a join. The app measures the latency of backend operations and uses that as the item's cost. In our experiment, the cache can hold 30k objects, and we drive the app with 1M requests sampled from a Zipfian distribution ($\alpha \approx 1$). When using normal HC, we measured a throughput of 5.0 kreq/s and a miss rate of



(a) Throughput measured over 30 second windows.



(b) Tail latency measured over 30 second windows.

Figure 12: Performance of NodeJS app fetching items from two different PSQL servers using HC with per-item and per-class costs. After 2 minutes, one PSQL server is stressed and takes longer to fetch items. The cache holds 30k objects and requests are Zipfian ($\alpha \approx 1$).

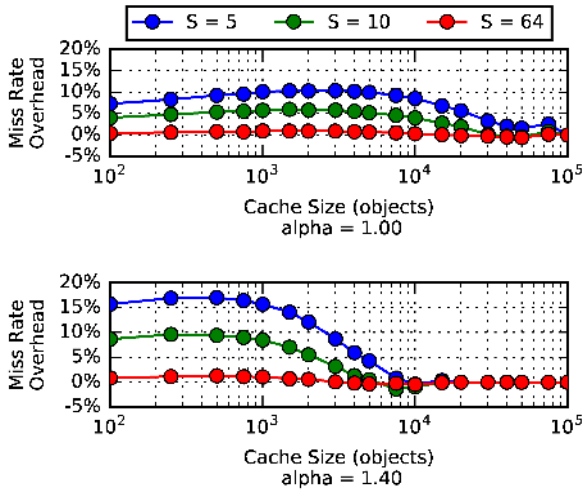


Figure 13: Simulated performance of HC for different sampling sizes compared to finding the true minimum. The request workloads are Zipfian distributions with different skew parameters.

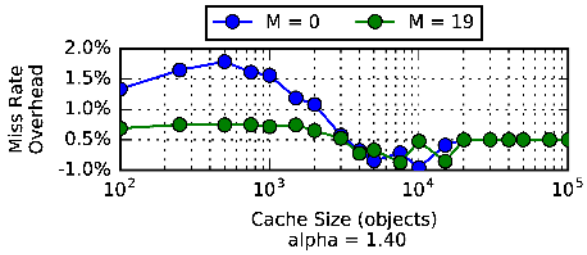


Figure 14: Simulation of HC using sampling technique that retains M items [42] on a Zipfian workload with $\alpha \approx 1.4$, compared to the performance of finding the true minimum.

0.11. When using HC-Cost, the miss rate was 0.17, which is 57% *higher*, but the throughput was 9.4 kreq/s, an 85% improvement over HC. HC-Cost traded off miss rate for lower overall cost, increasing overall performance.

Responding to backend load with classes. To demonstrate how cost classes can be used to deal with backend load, we designed a NodeJS application which performs key lookups on one of two different PSQL servers. The application measures the latency of the backend operation and uses that as the cost in our Redis prototype. Additionally, it sets the class of each cached object to indicate which backend served the object. This way, HC-Class will use a per-class cost estimate (exponentially WMA) when deciding which items to evict, rather than per-item. We evaluate the application by driving it with requests and measuring throughput and tail latency (Fig. 12). Two minutes into our test, we stress one PSQL backend using the Unix script `stress`. When one backend is loaded, throughput decreases and tail latency increases. By using per-class costs, HC-Class quickly adjusts to one class being more costly. With per-item costs, however, HC-Cost is only able to update the costs of items when they are (re)inserted. As a result, HC-Cost needs more time to settle to steady state performance as item costs are slowly updated to their correct values.

5.3 Accuracy of random sampling

Our eviction strategy's sampling impacts its miss-rate. Prior work [42] has studied the impact of this sampling in detail. Using order statistics [17], one can easily show that the expected rank of an evicted item is $n/(S+1)$, where n is the number of items in the cache and S is the sample size. For example, a cache of $n = 10$k items and a sample of $S = 64$ would evict the 154th lowest item on average. In practice we found that this loss of accuracy is not problematic. Specifically, we measured and compared the miss rate curves for varying sample sizes on two different popularity skews (Fig. 13). While the smoothness of the priority distribution impacts this accuracy—and extensions like expiration may introduce jaggedness into priorities—the dominating factor is how heavy the tail is and the likelihood of sampling an item from it. Sampling performs worse on the lighter-tailed distribution because there are fewer tail items in the cache, making them less likely to be sampled. However, for the sample size we use ($S = 64$), the performance gap relative to full accuracy is slight. Although this varies depending on the workload and cache-size, a sample of 64 items was large enough in all of our experiments, so the additional improvement of better sampling techniques would be limited. Further increasing the

sample size is not without cost: each sampled item's priority must be evaluated, which could become expensive depending on the complexity of the priority function.

Psounis and Prabhakar [42] proposed an optimization to random sampling that retains some number of samples between evictions. This can boost the accuracy of random sampling, however in our tests we found the miss rate benefits to be minimal. On the light-tail distribution (Fig. 14), we compare performance to the suggested settings of their technique. While performance does improve for smaller caches, the benefits are more limited as cache size increases. We believe this is because tail items in a large cache tend to be new items that are less likely to be retained from prior evictions, though a more in-depth analysis is needed to confirm this. As the benefits are limited (and parameters are sensitive to cache size and workload), we did not use this optimization.

6 Related Work

Our introduction and subsequent discussions survey the landscape of caching work, including recency-based approaches (e.g., [16, 41, 53]), frequency-based or hybrid approaches (e.g., [34,37]), marking algorithms and partial orderings (e.g., [16, 22]), and function-based approaches (e.g., [2, 46, 52]). All of these approaches rely on data structures and thus cannot achieve the flexibility and extensibility of hyperbolic caching.

Consider the approaches that improve recency caching by using multiple queues to incorporate some frequency measures into eviction. LRU-K [41] stores items in k queues and evicts based on the k-th most recent access. Other works employing multiple queues include 2Q [30], MQ [55], and LIRS [29]. ARC [37] automatically tunes the queue sizes of an LRU-2-like configuration. Several of these algorithms incorporate *ghost caches*, which track information about items no longer in the cache. (This technique could also be applied to hyperbolic caching, but we focused our work on caches that store information about items residing in the cache, as most production caches do.) All of these strategies incorporate frequency to balance the downsides of LRU. However, they are difficult to adapt to handle costs or other factors, due to their use of time-of-access metrics and priority orderings.

GreedyDual [53] exemplifies this difficulty because it attempts to incorporate cost into LRU, requiring a redesign. Cao and Irani [11] implemented GreedyDual using priority queues for size-aware caching in web proxies, and GDWheel [35] implemented GreedyDual in Memcached using a more efficient wheel data structure. The RIPQ system uses size awareness in a flash-based caching system [49]. Other cost-aware strategies have incorporated properties such as freshness (e.g., [46]), which is similar to expiration times but not as strict. In contrast to these approaches, a priority function based on frequency can easily adopt cost, expiration, or other factors.

Hyperbolic caching learns from the above and adopts a function-based approach based on frequency. The GDSF [13] work incorporates frequency into their priority function, while Yang and Zhang [52] use a priority function that is also similar to ours. However, these strategies build their solution on GreedyDual by setting an item's cost equal to its priority. In our tests, we found that the interaction between GreedyDual's priority queue and this frequency led to poor performance (3-4x the miss rate of LRU). Moreover, using a queue forces these strategies to "freeze" an item's priority once it enters the structure; in contrast, our priorities evolve continuously and freely.

Recent work in the systems community has looked at other aspects of caching that we do not address, such as optimizing memory overheads [19, 21], multi-tenant caching [14, 43], balancing memory slabs [14], cache admission [19], and reducing flash erasures when using flash storage [12, 36, 49]. Hyperbolic caching does not require memory for ordering data structures, but uses space to store the metadata used to compute item priorities. We have not studied allocation across multiple caches, but note that our framework obviates the need for separately tuned caches in some cases, e.g., by using our cost class extension to manage the pools of caches described in [40].

7 Conclusion

We have presented the design and implementation of hyperbolic caching. Our work combines theoretical insights with a practical framework that enables innovative, flexible caching. Notably, the priority function we use reorders items continuously along hyperbolic curves. We implemented our work in Redis and Django and applied it to a variety of real applications and systems. By using different extensions, we are able to match or exceed the performance of one-off caching solutions. A deeper analysis of the described extensions, such as for cost classes and expiration times, is part of our future work.

Acknowledgments. This work was supported by NSF CAREER Award #0953197. Part of this work was conducted during an internship at MSR NYC. We thank Amit Levi for access to Memcachier traces and Asaf Cidon for his help in obtaining them. We thank Siddhartha Jayanti for his assistance with theoretical analyses. Muthu Muthukrishnan coined the name "hyperbolic caching". Finally, we thank our shepherd, Rachit Agarwal.

References

[1] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, S. Sen, and A. Slivkins. A multiworld testing decision service. *CoRR*, abs/1606.03966, 2016.

[2] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the world wide web. *IEEE Trans. Knowledge and Data Eng.*, 11(1):95–107, 1999.

[3] S. Albers, L. M. Favrholdt, and O. Giel. On paging with locality of reference. *J. Comput. Syst. Sci*, 70(2):145–175, 2005.

[4] Amazon ElastiCache. `http://aws.amazon.com/elasticache`.

[5] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the Facebook social graph. In *SIGMOD*, 2013.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[7] Azure Redis Cache. `https://azure.microsoft.com/en-us/services/cache`.

[8] A. Balamash and M. Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys and Tutorials*, 6(1-4):44–56, 2004.

[9] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *J. Comput. System Sci.*, 50(2):244–258, 1995.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[11] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. Symposium on Internet Technologies and Systems (USITS)*, 1997.

[12] Y. Cheng, F. Douglis, P. Shilane, M. Trachtman, G. Wallace, P. Desnoyers, and K. Li. Erasing belady's limitations: In search of flash cache offline optimality. In *USENIX ATC*, 2016.

[13] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *Conf. on High-Performance Computing and Net.*, 2001.

[14] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, 2010.

[16] F. J. Corbato. A paging experiment with the multics system. In Feshbach and Ingard, editors, *In Honor of Philip M. Morse*, pages 217–228. MIT Press, 1969.

[17] H. A. David and H. N. Nagaraja. *Order Statistics*. Wiley Series in Probability and Statistics, 2003.

[18] Django. `https://www.djangoproject.com`.

[19] G. Einziger and R. Friedman. TinyLFU: A highly efficient cache admission policy. In *Proc. Parallel, Dist. and Net. Processing*, 2014.

[20] J. B. Estoup. Gammes stenographiques., 1916.

[21] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *NSDI*, 2013.

[22] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[23] B. Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004(124):5–5, 2004.

[24] P. A. Franaszek and T. J. Wagner. Some distribution-free aspects of paging algorithm performance. *J. ACM*, 21(1):31–39, 1974.

[25] S. Goel, A. Anderson, J. M. Hofman, and D. J. Watts. The structural virality of online diffusion. *Management Science*, 62(1):180–196, 2016.

[26] Guava: Google Core Libraries for Java. `https://github.com/google/guava`.

[27] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[28] Hyperbolic caching. `https://github.com/kantai/hyperbolic-caching`.

[29] S. Jiang and X. Zhang. LIRS: an efficient low interreference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[30] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[31] H. Kaplan, R. E. Tarjan, and K. Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *SODA*, pages 836–844, 2001.

[32] A. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. *SIAM J. Computing*, 30(3):906–922, 2000.

[33] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM J. Computing*, 30(1):300–317, 2000.

[34] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.*, 50(12):1352–1361, 2001.

[35] C. Li and A. L. Cox. GD-Wheel: A cost-aware replacement policy for key-value stores. In *EUROSYS*, 2015.

[36] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, 2015.

[37] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[38] Memcachier. `http://www.memcachier.com`.

[39] Multiworld Testing Decision Service. `http://aka.ms/mwt`.

[40] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[41] E. J. O'neil, P. E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Record*, 22(2):297–306, 1993.

[42] K. Psounis and B. Prabhakar. Efficient randomized web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Trans. Networking*, 10(4):441–455, 2002.

[43] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI*, pages 393–406, 2016.

[44] Redis Key-Value Store. `http://http://redis.io`.

[45] Using Redis as an LRU Cache. `https://redis.io/topics/lru-cache#approximated-lru-algorithm`.

[46] J. Shim, P. Scheuermann, and R. Vingralek. Proxy cache algorithms: Design, implementation, and performance. *IEEE Trans. Knowledge and Data Eng.*, 11(4):549–562, 1999.

[47] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[48] Storage Performance Council Trace Repository. `http://www.storageperformance.org/specs/#traces`, 2002.

[49] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *FAST*, 2015.

[50] Varnish HTTP Cache. `https://www.varnish-cache.org`.

[51] ViralSearch: Identifying and Visualizing Viral Content. `https://www.microsoft.com/en-us/research/video/viralsearch-identifying-and-visualizing-viral-content/`.

[52] Q. Yang and H. Zhang. Taylor series prediction: A cache replacement policy based on second-order trend analysis. In *HICSS*. IEEE, 2001.

[53] N. Young. *Competitive paging and dual-guided on-line weighted caching and matching algorithms*. PhD thesis, Princeton University, 1991.

[54] N. E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

[55] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, 2001.

[56] G. K. Zipf. *Selected studies of the Principle of Relative Frequency in Language.* Harvard Univ. Press, 1932.